Лекция 7:

**Azure Services Platform**

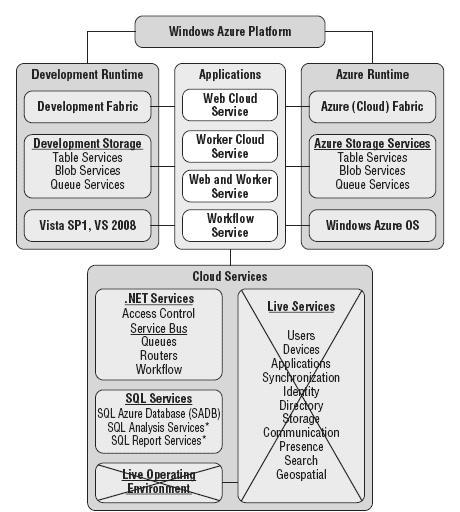
Цель данной лекции – получить представление об архитектуре Windows Azure

### Архитектура Windows Azure Platform

*Платформа Windows Azure* – это модель Платформа как Сервис, которая предполагает запуск приложений на серверах и связанной сетевой инфраструктуре, размещенной в центрах обработки данных Microsoft и имеющей доступ в Интернет. Платформа состоит из мастшабируемой "облачной" операционной системы, фабрики хранения данных и связанных сервисов доставки через физические или логические (виртуализация) экземпляры Windows Server 2008. Комплект средств разработки Windows Azure (SDK) обеспечивает разработку версии "облачных сервисов", также хорошо, как инструменты и интерфейсы прикладного программирования (API), необходимые для разработки, разворачивания и управления масштабируемых сервисов в Windows Azure, включая шаблоны приложений Azure для Visual Studio 2008 и 2010. На [рисунке 6.1](http://www.intuit.ru/studies/courses/673/529/lecture/11921?page=1#image.6.1) изображены компоненты основа "облачной" платформы и компоненты разработчика.

Согласно Microsoft, при использовании Azure Вы получаете:

* Адаптация существующих приложений для работы с веб сервисами;
* Построение, изменение и распределение приложений в Сети с минимальными локальными ресурсами;
* Выполняют услуги, таких как хранение больших объемов данных, пакетная обработка данных, вычисления больших объемов данных, и так далее;
* Создание, тестирование, отладка и распределение веб сервисов быстро и недорого;
* Снижение стоимости и рисков построения и распространения местных ресурсов;
* Снижение затрат и усилий на ИТ управление;

****

**Рис. 6.1.** Компоненты платформы Windows Azure и Комплекта средств разработки

Экономическая обстановка Microsoft во время выпуска платформы Azure акцентировала внимание на снижении затрат, что является основным поводом для использования Azure мелкими, средними и крупными ИТ отделами.

Microsoft разработали платформу Azure, позволив .NET разработчикам усилить их опыт создания в Visual Studio 2008 (и выше) ASP.NET веб приложений и Windows Communication Framework (WCF) сервисов. Проекты веб приложений запускаются в изолированной версии Internet Information Services (IIS) 7. Веб приложения и веб сервисы запускаются в частично доверенном механизме защиты, позволяющем ограничивать доступ коду к ресурсам компьютера (Code Access Security), который приблизительно соответсвует среднему уровню доверия ASP.NET и ограничивает доступ к некоторым ресурсам операционной системы. Комплект средств разработки Windows Azure (Март 2009) позволяет использоваться полный доступ к ресурсам компьютера для запуска не .NET кода, использования .NET библиотек, которые требуют полного доверия и процесс обработки взаимодействия, используя программные каналы (Pipe). Microsoft обещает поддержку запуска программного кода Ruby, PHP и Python в "облачной" платформе. Исходный вариант платформы разработки был ограничен средой программирования Visual Studio 2008 и выше с планом на поддержку инструментов Eclipse. Платформа Windows Azure поддерживает веб стандарты и протоколы включая SOAP, HTTP, XML, Atom и AtomPub.

Исходной точкой входа для разработчиков Azure для размещения ASP.NET приложений в облако является портал Windows Azure по адресу https://windows.azure.com/Cloud/Provisioning/Default.aspx. Портал требует входа с использованием Windows Live ID. Предварительная версия Azure (Community Technical Previews , CTPs) требует разных токенов для:

Windows Azure, которая включает:

* Azure Hosted service;
* Storage Accounts;

SQL Azure Live службы, которые включают:

* Live Framework: Предварительная версия;
* Live Services: существующие интерфейсы прикладного программирования.

Live сервисы: существующие интерфейсы прикладного программирования не являются частью предварительной версии и не требуют токена. С начала 2009 года Windows Azure токен дает право на одну учетную запись Hosted Service, две учетные записи для Storage. Вы запрашиваете токены Azure через страницу Microsoft Connect, на которую можно попасть со страницы портала.

На рисунке 6.1 показана страница учетной записи, которая содержит ссылки на страницы настроек и управления SQL Azure, сервисами .NET и Live сервисами. Эта страница также позволяет вам получать Azure и Live Framework CTP токены. Страница Live Alerts позволяет настроить, как и когда получать сообщения, содержащие критические сигналы приложения, информационные бюллетени и обновления портала.

Предварительная версия марта 2009 года включает геолокацию, которая позволяет владельцам учетных записей выбрать центры обработки данных для расположения Hosted Services и Storage Accounts. Например, USA-Northwest (Quincy, WA) и USA-Southeast (San Antonio, TX.). Вы можете добавить наборы Hosted Services и Storage Accounts в группу, чтобы гарантировать, что сервисы и хранилище располагаются в одном и том же центре обработки данных, для того, чтобы увеличить производительность.

Вы можете вставить GUID полученные в сообщении от члена команды Azure в текстовое окно Resource Token ID и нажать Claim Token для добавление еще одной или больше точки входа для подходящего объекта или нескольких объектов в список.

Если вы хотите разрабатывать веб сайты с поддержкой Live Framework или Mesh (программный комплекс для синхронизации данных в кроссплатформенных средах разработанный компанией Microsoft) веб приложений, необходимо запросить токен Live Framework по email meshctpe@microsoft.com. После того, как вы получите Live Framework токен, вы можете скачать и установить текущие версии Live Framework SDK и Live Framework Tools для Visual Studio по ссылкам указанным на странице <http://dev.live.com/liveframework/sdk/>. Вы должны оплатить Live Framework токен для того, чтобы скачать Live SDK и дополнительные инструменты. Нет необходимости использовать учетную запись Windows Azure для тестирования Azure Hosted Services and Storage Services, потому что платформа разработки Azure эмулирует "облачные" сервисы Azure на вашем компьютере.

### Windows Azure Storage

*Хранилище Windows Azure Storage* обеспечивает разработчикам возможность хранения данных в облаке. Приложение может выполнять доступ к своим данным в любой момент времени из любой точки планеты, хранить любой объем данных и как угодно долго. При этом данные гарантированно не будут повреждены и утеряны. Windows Azure Storage предлагает богатый набор абстракций данных:

* Windows Azure Table – обеспечивает структурированное хранилище состояний сервиса.
* Windows Azure Blob – обеспечивает хранилище больших элементов данных.
* Windows Azure Queue – обеспечивает диспетчеризацию асинхронных заданий для реализации обмена данными между сервисами.

#### Azure Table Services

*Windows Azure Table* - структурированное хранилище, которе поддерживает высокомасштабируемые таблицы в облаке, которые могут содержать миллиарды сущностей и терабайты данных. По мере увеличения трафика, система будет эффективно масштабироваться, автоматически подключая тысячи серверов. Структурированное хранилище реализовано в виде таблиц (Tables), в которых располагаются сущности (Entities), содержащие ряд именованных свойств (Properties). Вот некоторые из основных характеристик Windows Azure Table:

* Поддержка LINQ, ADO .NET Data Services и REST.
* Контроль типов во время компиляции при использовании клиентской библиотеки ADO .NET Data Services.
* Богатый набор типов данных для значений свойств.
* Поддержка неограниченного количества таблиц и сущностей без ограничения размеров таблиц.
* Поддержка целостности для каждой сущности.
* Нежесткая блокировка при обновлениях и удалениях.
* Для запросов, выполнение которых требует длительного периода времени, или запросов, прерванных по завершению времени ожидания, возвращаются частичные результаты и маркер продолжения

Рассмотрим модель данных таблицы Windows Azure Table:

* **Учетная запись хранилища (Storage Account)** – для доступа к Windows Azure Storage приложение должно использовать действительную учетную запись. Новую учетную запись можно создать через веб-интерфейс портала Windows Azure. Как только учетная запись создана, пользователь получает 256-разрядный секретный ключ, который впоследствии используется для аутентификации запросов этого пользователя к системе хранения. В частности, с помощью этого секретного ключа создается подпись HMAC SHA256 для запроса. Эта подпись передается с каждым запросом данного пользователя для обеспечения аутентификации. Имя учетной записи входит в состав имени хоста в URL. Для доступа к таблицам используется следующий формат имени хоста: <имяУчетнойЗаписи>.table.core.windows.net.
* **Таблица (Table)** – содержит набор сущностей. Область действия имен таблиц ограничена учетной записью. Приложение может создавать множество таблиц в рамках учетной записи хранилища.
* **Сущность (строка) (Entity (Row))** – Сущности (сущность является аналогом "строки") – это основные элементы данных, хранящиеся в таблице. Сущность включает набор свойств. В каждой таблице имеется два свойства, которые образуют уникальный ключ для сущности.
* **Свойство (столбец) (Property (Column))** – Представляет отдельное значение сущности. Имена свойств чувствительны к регистру. Для значений свойств поддерживается богатый набор типов.
* **Ключ секции (PartitionKey)** – Первое свойство ключа каждой таблицы. Эта система использует данный ключ для автоматического распределения сущностей таблицы по множеству узлов хранения.
* **Ключ строки (RowKey)** – Второе свойство ключа таблицы. Это уникальный ID сущности в рамках секции. PartitionKey в сочетании с RowKey уникально идентифицирует сущность в таблице.
* **Временная метка (Timestamp)** – Каждая сущность имеет версию, сохраняемую системой.
* **Секция (Partition)** – Набор сущностей в таблице с одинаковым значением ключа секции.
* **Порядок сортировки (Sort Order)** – Для CTP-версии предоставляется всего один индекс, в котором все сущности сортированы по PartitionKey и затем по RowKey. Это означает, что запросы с указанием этих ключей будут более эффективными, и все возвращаемые результаты будут сортированы по PartitionKey

Таблица имеет гибкую схему. Windows Azure Table отслеживает имя и типизированное значение каждого свойства каждой сущности. Приложение может моделировать фиксированную схему на стороне клиента, обеспечивая одинаковый набор свойств для всех создаваемых сущностей.

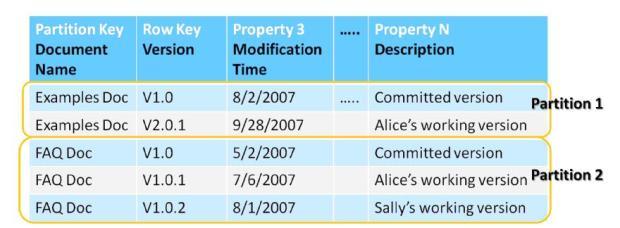
Рассмотрим некоторые дополнительные сведения о сущностях:

* Сущность может иметь до 255 свойств, включая обязательные системные свойства: PartitionKey, RowKey и Timestamp. Имена всех остальных свойств сущностей определяются приложением.
* Свойства PartitionKey и RowKey строкового типа.
* Свойство Timestamp является доступным только для чтения обслуживаемым системой свойством, которое должно рассматриваться как непрозрачное свойство.
* Отсутствие фиксированной схемы – Windows Azure Table не сохраняет никакой схемы, поэтому все свойства хранятся как пары <имя, типизированное значение>. Это означает, что свойства сущностей одной таблицы могут сильно отличаться. В таблице даже может быть две сущности, свойства которых имеют одинаковые имена, но разные типы значений.
* Суммарный объем всех данных сущности не может превышать 1 МБ. Сюда входит размер имен свойств, а также размер значений свойств или их типов, включая и два обязательных свойства ключей (PartitionKey и RowKey).
* Поддерживаются типы Binary, Bool, DateTime, Double, GUID, Int, Int64, String. Ограничения представлены в таблице ниже.

|  |  |
| --- | --- |
| Таблица 6.1. | |
| **Тип свойства** | **Описание** |
| Binary | Массив байтов размером до 64 КБ. |
| Bool | Булево значение. |
| DateTime | 64-разрядное значение, представляющее время в формате UTC. Поддерживаемый диапазон значений: от 1/1/1600 до 12/31/9999. |
| Double | 64-разрядное значение с плавающей точкой. |
| GUID | 128-разрядный глобально уникальный идентификатор. |
| Int | 32-разрядное целое значение. |
| Int64 | 64-разрядное целое значение. |
| String | 16-разрядное UTF-кодированное значение. Размер строковых значений может быть до 64 КБ. |

Windows Azure Table обеспечивает возможность масштабирования таблиц до тысяч узлов хранения через распределение сущностей в таблице. При распределении сущностей желательно обеспечить, чтобы сущности, входящие в одно множество, располагались в одном узле хранения. Приложение формирует эти множества соответственно значениям свойства PartitionKey сущностей.

Приложениям должна быть известна рабочая нагрузка каждой отдельно взятой секции. Для обеспечения желаемых результатов тестирование должно моделировать максимальную рабочую нагрузку.



**Рис. 6.2.** Примеры секций

На рисунке выше представлена таблица, содержащая множество версий документов. Каждая сущность данной таблицы соответствует определенной версии определенного документа. В этом примере ключом секции таблицы является имя документа, и ключом строки – номер версии. Имя документа и версия уникально идентифицируют каждую сущность таблицы. В данном примере секцию образуют все версии одного документа.

Хорошая масштабируемость системы хранения достигается за счет распределения секций по множеству узлов хранения.

Система отслеживает характер использования секций и автоматически равномерно распределяет эти секции по всем узлам хранения. Это позволяет системе и приложению масштабироваться соответственно количеству запросов к таблице. То есть если некоторые секции запрашиваются больше других, система автоматически разнесет их на несколько узлов хранения, таким образом, распределяя трафик между множеством серверов. Однако секция, т.е. все сущности, имеющие одинаковый ключ секции, будут обслуживаться как один узел. Но даже несмотря на это, объем данных в рамках секции не ограничен емкостью хранилища одного узла хранения.

Сущности одной секции хранятся вместе. Это обеспечивает наиболее эффективную обработку запросов к секции. Более того, в этом случае приложение может использовать все преимущества эффективного кэширования и других оптимизаций производительности, обеспечиваемых расположением данных в секции.

В примере выше секцию образуют все версии одного документа. Таким образом, для извлечения всех версий данного документа необходимо выполнить доступ всего к одной секции. Чтобы получить все версии документов, измененные до 5/30/2007, придется запрашивать несколько секций, что будет не так эффективно и более ресурсоемко, поскольку по запросу должны будут проверяться все секции, которые к тому же могут располагаться на разных узлах хранения.

Выбор ключа секции важен с точки зрения обеспечения эффективного масштабирования приложения. При этом необходимо найти компромисс между размещением сущностей в одной секции, что обеспечивает большую эффективность запросов, и масштабируемостью таблицы, поскольку, чем больше секций в таблице, тем проще для Windows Azure Table распределить нагрузку между множеством серверов.

Для наиболее частых и критичных по времени ожидания запросов PartitionKey должен быть включен как часть выражения запроса. Запрос, в котором указан PartitionKey, будет намного эффективнее, поскольку в этом случае просматриваются сущности только одной секции. Если при выполнении запроса PartitionKey не указан, в поисках необходимых сущностей просматриваются все секции таблицы, что значительно снижает эффективность.

Далее представлены некоторые советы и рекомендации по выбору PartitionKey для таблицы:

1. Прежде всего, выявите важные свойства таблицы. Это свойства, используемые в условиях запросов.
2. Из этих важных свойств выберите потенциальные ключи.
   * Из преобладающего запроса выберите свойства, используемые в условиях. Важно понять, какой запрос будет преобладающим для приложения.
   * Это будет исходный набор свойств ключей.
   * Расставьте свойства ключей в порядке их значимости в запросе.
3. Проверьте, обеспечивают ли свойства ключей уникальную идентификацию сущности? Если нет, включите в набор ключей уникальный идентификатор.
4. Если имеется только одно свойство ключа, используйте его в качестве PartitionKey.
5. Если имеется только два свойства ключей, первое используйте как ParitionKey и второе – как RowKey.
6. При наличии более двух свойств ключей можно попытаться распределить их в две группы: первая группа будет PartitionKey, и вторая – RowKey. При таком подходе приложение должно знать, что PartitionKey, например, состоит из двух ключей, разделенных "-".

Теперь, когда приложение имеет набор потенциальных ключей, необходимо убедиться, что выбранная схема секционирования является масштабируемой:

1. Исходя из статистических данных интенсивности использования приложения, определите, не приведет ли секционирование по выбранному выше PartitionKey к созданию слишком загруженных секций, которые не смогут эффективно обслуживаться одним сервером? Проверить это можно, применив нагрузочное тестирование секции таблицы. Для этого в тестовой таблице создается секция соответственно выбранным ключам. Она подвергается пиковой нагрузке, полученной исходя из предполагаемых полезной нагрузки и запросов. Это позволяет проверить, может ли секция таблицы обеспечить необходимую производительность приложения.
2. Если секция таблицы проходит нагрузочное тестирование, ключи выбраны правильно.
3. Если секция таблицы не проходит нагрузочного тестирования, найдите ключ секции, который обеспечил бы более узкое подразделение сущностей. Это можно сделать через объединение выбранного ключа секции со следующим свойством ключа, или выбрав в качестве ключа секции другое важное свойство. Целью этой операции должно быть создание большего количества секций, чтобы не возникало одной слишком большой или слишком загруженной секции.
4. Система спроектирована так, что обеспечивает необходимое масштабирование и обработку большого количества запросов. Но при чрезвычайно высокой интенсивности запросов ей приходится выполнять балансировку нагрузки, в результате чего некоторые из запросов могут завершаться ошибкой превышения времени ожидания. Сократить или устранить ошибки такого рода может снижение интенсивности запросов. Вообще говоря, такие ошибки возникают редко; однако если вы столкнулись с частыми или неожиданными ошибками превышения времени ожидания, свяжитесь с нами через сайт

MSDN, мы обсудим, как оптимизировать использование Windows Azure Table и предотвратить возникновение таких ошибок в вашем приложении.

Также можно проанализировать расширяемость выбранных ключей, особенно если на момент их выбора нет точных сведений о характеристиках пользовательского трафика. В этом случае важно выбирать ключи, которые можно легко расширять для обеспечения более тонкого секционирования. Далее в данном документе приводится подробный пример этого.

Для таблиц и сущностей поддерживаются следующие базовые операции:

* Создание таблицы или сущности.
* Извлечение таблицы или сущности с применением фильтров.
* Обновление сущности (но не таблицы).
* Удаление таблицы или сущности.

Для работы с таблицами в .NET-приложении можно просто использовать ADO.NET Data Services.

В следующей таблице приведен список предлагаемых API. Поскольку применение ADO.NET Data Services в итоге сводится к передаче REST-пакетов, приложения могут использовать REST напрямую. Кроме того, что REST обеспечивает возможность доступа к хранилищу посредством не-.NET языков, он также позволяет реализовывать более тонкое управление сериализацией/десериализацией сущностей, что пригодится при работе с такими сценариями, как наличие разных типов сущностей или более чем 255 свойств в таблице и т.д.

##### Пример

В приведенных ниже примерах описываются операции с таблицей "Blogs". В этой таблице хранятся блоги для приложения MicroBlogging.

В приложении MicroBlogging есть две таблицы: Channels (Каналы) и Blogs (Блоги). Имеется список каналов, блоги публикуются в определенном канале. Пользователи подписываются на каналы и ежедневно получают новые блоги этих каналов.

В данном примере рассмотрим только таблицу Blogs и приведем примеры следующих операций с ней:

1. Описание схемы таблицы
2. Создание таблицы
3. Вставка блога в таблицу
4. Получение списка блогов из таблицы
5. Обновление блога в таблице
6. Удаление блога из таблицы

Схема таблицы описывается как C#-класс. Такую модель использует ADO.NET Data Services. Схема известна только клиентскому приложению и упрощает доступ к данным. Сервер схему не применяет.

Рассмотрим описание сущностей Blog, хранящихся в таблице Blogs. Каждая сущность блога содержит следующие данные:

1. Имя канала (ChannelName) – канал, в котором размещается блог.
2. Дата размещения.
3. Текст (Text) – содержимое тела блога.
4. Рейтинг (Rating) – популярность этого блога.

Во-первых, обратите внимание, что для таблицы определен PartitionKey, представляющий имя канала, частью которого является блог, и в качестве RowKey используется дата размещения блога. PartitionKey и RowKey – ключи таблицы Blogs, они объявляются посредством атрибута класса DataServiceKey (Ключ сервиса данных). То есть таблица Blogs секционирована по именам каналов (ChannelName). Это позволяет приложению эффективно извлекать самые недавние блоги канала, на который подписан пользователь. Кроме ключей, в качестве свойств объявлены характерные для пользователя атрибуты. Все свойства имеют открытые (public) методы считывания и присвоения значения и хранятся в таблице Windows Azure Table. Итак, в примере ниже:

* Text и Rating хранятся для экземпляра сущности в таблице Azure.
* RatingAsString нет, потому что для него не определен метод присвоения значения.
* Id не хранится, потому что методы доступа не public.

[DataServiceKey("PartitionKey", "RowKey")]

public class Blog

{

// ChannelName

public string PartitionKey { get; set; }

// PostedDate

public string RowKey { get; set; }

// Определяемые пользователем свойства

public string Text { get; set; }

public int Rating { get; set; }

public string RatingAsString { get; }

protected string Id { get; set; }

}

Далее рассмотрим, как создать таблицу Blogs для учетной записи хранилища. Создание таблицы аналогично созданию сущности в основной таблице "Tables". Эта основная таблица определена для каждой учетной записи хранилища, и имя каждой таблицы, используемой учетной записью хранения, должно быть зарегистрировано в основной таблице. Описание класса основной таблицы приведено ниже, где свойство TableName (Имя таблицы) представляет имя создаваемой таблицы.

[DataServiceKey("TableName")]

public class TableStorageTable

{

public string TableName { get; set; }

}

Фактическое создание таблицы происходит следующим образом:

// Uri сервиса: "http://<Account>.table.core.windows.net/"

DataServiceContext context = new DataServiceContext(serviceUri);

TableStorageTable table = new TableStorageTable("Blogs");

// Создаем новую таблицу, добавляя новую сущность

// в основную таблицу "Tables"

context.AddObject("Tables", table);

// результатом вызова SaveChanges является отклик сервера

DataServiceResponse response = context.SaveChanges();

serviceUri – это uri сервиса таблицы, http://<Здесь указывается имя учетной записи>.table.core.windows.net/. DataServiceContext (Контекст сервиса данных) – один из основных классов сервиса данных ADO.NET, представляющий контекст времени выполнения для сервиса. Он обеспечивает API для вставки, обновления, удаления и запроса сущностей с помощью либо LINQ, либо RESTful URI и сохраняет состояние на стороне клиента.

Рассмотрим вставку элемента Blog. Чтобы вставить сущность, приложение должно выполнить следующее.

1. Создать новый C#-объект и задать все свойства.
2. Создать экземпляр DataServiceContext, который представляет подключение к серверу в сервисе данных ADO .NET для вашей учетной записи хранилища.
3. Добавить C#-объект в контекст.
4. Вызвать метод SaveChanges (Сохранить изменения) объекта DataServiceContext для отправки запроса серверу. Это обеспечивает отправку на сервер HTTP-запроса с сущностью в XML-формате ATOM.

Далее представлены примеры кода для перечисленных выше операций:

Blog blog = new Blog {

PartitionKey = "Channel9", // ChannelName

RowKey = DateTime.UtcNow.ToString(), // PostedDate

Text = "Hello",

Rating = 3

};

serviceUri = new Uri("http://<account>.table.core.windows.net");

var context = new DataServiceContext(serviceUri);

context.AddObject("Blogs", blog);

DataServiceContext response = context.SaveChanges();

Запрос сущностей выполняется с помощью встроенного в C# языка запросов LINQ (Language Integrated Query). В данном примере извлечем все блоги, рейтинг которых равен 3.

При обработке запроса (например, с помощью выражение foreach), он передается на сервер. Сервер отправляет результаты в XML-формате ATOM. Клиентская библиотека ADO .NET Data Services десериализует результаты в C#-объекты, после чего они могут использоваться приложением.

var serviceUri = new Uri("http://<account>.table.core.windows.net");

DataServiceContext context = new DataServiceContext(serviceUri);

// LINQ-запрос с использованием DataServiceContext для выбора

// из таблицы Blogs всех сущностей блогов, для которых rating = 3

var blogs =

from blog in context.CreateQuery<blog>("Blogs")

where blogs.Rating == 3

select blog;

// запрос отправляется на сервер и выполняется

foreach (Blog blog in blogs) { }

Обновление сущности выполняется следующим образом.

1. Создается DataContext (Контекст данных), свойству MergeOption (Вариант объединения) которого задается значение OverwriteChanges (Перезапись изменений) или PreserveChanges (Сохранение изменений), как описывается в разделе 4.8. Это обеспечивает правильную обработку ETag для каждого извлекаемого объекта.
2. С помощью LINQ DataContext получает сущность, которая будет обновляться. Извлечение ее с сервера гарантирует обновление ETag в сущностях, отслеживаемых контекстом, и то, что при последующих обновлениях и удалениях в заголовке if-match будет использоваться обновленный ETag. Меняем C#-объект, представляющий сущность.
3. Возвращаем C#-объект в тот же DataContext для обновления. Использование того же DataContext гарантирует автоматическое повторное использование ETag, полученного ранее для этого объекта.
4. Вызываем метод SaveChanges для отправки запроса на сервер.

Blog blog =

(from blog in context.CreateQuery<blog>("Blogs")

where blog.PartitionKey == "Channel9"

&& blog.RowKey == "Oct-29"

select blog).FirstOrDefault();

blog.Text = "Hi there";

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

4.7 Удаление Blog

Удаление сущности аналогично ее обновлению. Для этого извлекаем сущность с помощью DataServiceContext и вызываем для содержимого вместо метода UpdateObject метод DeleteObject (Удалить объект).

// Получаем объект Blog для ("Channel9", "Oct-29")

context.DeleteObject(blog);

DataServiceResponse response = context.SaveChanges();

Рассмотрим рекомендации по работе с DataServiceContext:

* Объект DataServiceContext не обеспечивает безопасность потоков, поэтому он не может использоваться совместно разными потоками, а также имеет непродолжительное время существования.
* DataServiceContext не является объектом с длительным временем жизни. Вместо того, чтобы использовать один DataServiceContext в течение всей жизни потока, рекомендуется создавать объект DataServiceContext каждый раз, когда возникает необходимость выполнить ряд транзакций с WindowsAzureTable, и затем удалять этот объект.
* Если для всех вставок/обновлений/удалений используется один экземпляр DataServiceContext и возникает сбой при выполнении SaveChanges, сведения об операции, давшей сбой, сохраняются в DataServiceContext. При последующем вызове SaveChanges попытка выполнить эту операцию повторяется.
* DataServiceContext имеет свойство MergeOption, которое используется для управления тем, как DataServiceContext обрабатывает отслеживаемые сущности. Возможные значения:
  + AppendOnly (Только добавление): Это значение по умолчанию, при использовании которого DataServiceContext не загружает экземпляр сущности с сервера, если он уже имеется в его кэше.
  + OverwriteChanges: DataServiceContext всегда загружает экземпляр сущности с сервера и перезаписывает предыдущий вариант сущности, т.е. обеспечивает соответствие экземпляра сущности ее текущему состоянию.
  + PreserveChanges: Если экземпляр сущности существует в DataServiceContext, он не загружается из постоянного хранилища. Все изменения, вносимые в свойства объектов в DataServiceContext, сохраняются, но ETag обновляется, поэтому данную опцию следует использовать при необходимости восстановления после ошибок совместного доступа с нежесткой блокировкой.
  + NoTracking (Без отслеживания): DataServiceContext не отслеживает экземпляры сущностей. Обновление сущности в контексте без отслеживания реализуется с помощью Etag, который обновляется посредством AttachTo. Этот вариант не рекомендуется к применению.

context.AttachTo("Blogs", blog, "etag to use");

context.UpdateObject(blog);

context.SaveChanges();

Когда MergeOption контекста задано значение AppendOnly и объект DataServiceContext уже отслеживает сущность в результате предыдущей операции извлечения или добавления, повторное извлечение сущности с сервера не приведет к обновлению отслеживаемой сущности в контексте. Таким образом, если сущность на сервере была изменена, последующие обновления/удаления приведут к сбою необходимых условий ( PreCondition ). В примере кода раздела 5 MergeOption задано значение PreserveChanges, которое обеспечивает, что сущность будет загружаться с сервера всегда.

Результатом всех рассматриваемых выше операций является передача HTTP-сообщений на и с сервера. Приложение может отказаться от использования клиентской библиотеки .NET и работать на уровне HTTP/REST.

Рассмотрим параллельные обновления. Для обновления сущности необходимо выполнить следующие операции.

1. Получить сущность с сервера
2. Обновить объект локально и вернуть его на сервер.

Предположим, два процесса, выполняющихся параллельно, пытаются обновить одну и ту же сущность. Поскольку шаги 1 и 2 не являются неделимыми, на любом из них может возникнуть ситуация внесения изменений в уже устаревшую версию сущности. Для решения этой проблемы Windows Azure Table использует нежесткую блокировку.

1. Для каждой сущности система сохраняет версию, которая изменяется сервером при каждом обновлении.
2. При извлечении сущности, сервер отправляет эту версию клиенту в виде ETag HTTP.
3. Когда клиент передает запрос UPDATE (обновить) на сервер, он отправляет на него этот ETag в виде заголовка If-Match.
4. Если версия сущности, хранящаяся на сервере, аналогична ETag в заголовке If-Match, изменение принимается, и хранящаяся на сервере сущность получает новую версию. Эта новая версия возвращается клиенту как заголовок ETag.
5. Если версия сущности на сервере отличается от ETag в заголовке If-Match, изменение отклоняется, и клиенту возвращается HTTP-ошибка "precondition failed" (необходимое условие не выполнено).

При получении ошибки "precondition failed" типовым поведением клиентского приложения будет повторение всей операции, как показано в фрагменте кода ниже.

1. Приложение должно извлечь этот объект снова, т.е. получить его последнюю версию.
2. Обновить объект локально и вернуть его на сервер.

При использовании клиентской библиотеки .NET приложение получает HTTP-код ошибки в виде исключения DataServiceRequestException.

В примере ниже два разных клиента выполняют один и тот же код для изменения текста. Эти два клиента пытаются задать Text разные значения.

1. Они извлекают сущность. При этом для каждой сущности извлекается ETag, например, "v1". Оба клиента полагают, что предыдущая версия сущности – "v1".
2. Каждый клиент локально обновляет свойство Text.
3. Каждый клиент вызывает методы UpdateObject и SaveChanges.
4. Каждый клиент отправляет на сервер HTTP-запрос с заголовком "If-Match: v1".
5. Запрос одного из клиентов попадает на сервер первым.
   * Сервер сравнивает заголовок If-Match с версией сущности. Они совпадают.
   * Сервер применяет изменение.
   * Версия сущности на сервере обновляется и становится "v2".
   * В качестве ответа клиенту отправляется новый заголовок "ETag:v2".
6. Далее на сервер поступает запрос другого клиента. На этот момент изменения первого клиента уже применены.
   * Сервер сравнивает заголовок If-Match с версией сущности. Они не совпадают, поскольку версия сущности уже изменена на "v2", тогда как в запросе указывается версия "v1".
   * Сервер отклоняет запрос.

// Задаем такой вариант объединения, который обеспечивает

// сохранение обновлений, но позволяет обновление etag.

// По умолчанию применяется значение AppendOnly, при котором

// уже отслеживаемая сущность не перезаписывается значениями,

// полученными с сервера, в результате чего в случае изменения

// сущности на сервере используется недействительный etag.

context.MergeOption = MergeOption.PreserveChanges;

Blog blog =

(from blog in context.CreateQuery<blog>(&quot;Blogs&quot;)

where blog.PartitionKey == &quot;Channel9&quot;

&& blog.RowKey == &quot;Oct-29&quot;

select blog).FirstOrDefault();

blog.Text = &quot;Hi there again&quot;;

try

{

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

}

catch (DataServiceRequestException e)

{

OperationResponse response = e.Response.First();

if (response.StatusCode == (int)HttpStatusCode.PreconditionFailed)

{

// выполняем запрос объекта повторно, чтобы получить

// последний etag, и проводим обновление

}

}

Для безусловного обновления сущности приложение выполняет следующее:

1. Создает новый объект DataServiceContext или, в случае использования существующего контекста, отсоединяет объект, как демонстрирует пример ниже.
2. Присоединяем сущность к контексту и используем "\*" как новое значение ETag.
3. Обновляем сущность.
4. Вызываем SaveChanges.

// задаем опцию объединения, разрешающую перезапись,

// чтобы обеспечить возможность обновления отслеживаемой сущности

context.Detach(blog);

// Присоединяем сущность к контексту, используя имя таблицы, сущность,

// которая должна быть обновлена, и "\*" как значение etag.

context.AttachTo("Blogs", blog, "\*");

blog.Text = "Hi there again";

try

{

context.UpdateObject(blog);

DataServiceResponse response = context.SaveChanges();

}

catch (DataServiceRequestException e)

{

// Обработка ошибки, но в данном случае формировнание ошибки PreCondition невозможно

}

Для запросов, которые могут возвращать большое количество результатов, система обеспечивает два механизма:

1. Возможность получать первые N сущностей, используя LINQ-функцию Take(N).
2. Маркер продолжения, который обозначает место начала следующего множества результатов.

Система поддерживает функцию возвращения первых N соответствующих запросу сущностей. Например, если программа разрабатывается на .NET, для извлечения первых N сущностей (в данном примере это первые 100 сущностей) можно использовать LINQ-функцию Take(N).

serviceUri = new Uri("http://<account>.table.core.windows.net");

DataServiceContext svc = new DataServiceContext(serviceUri);

var allBlogs = context.CreateQuery<blog>("Blogs");

foreach (Blog blog in allBlogs.Take(100))

{

// выполняем некоторые операции с каждым блогом

}

Аналогичная функциональность поддерживается в интерфейсе REST через опцию строки запроса $top=N. Например, запрос "GET http://<UriСервиса>/Blogs?$top=100" обеспечил бы возвращение первых 100 сущностей, соответствующих запросу. Фильтрация выполняется на сервере, поэтому в ответе клиенту может быть передано максимум 100 сущностей.

1. В запросе указывается максимальное число сущностей, которое должно быть возвращено.
2. Количество сущностей превышает максимально разрешенное сервером число сущностей в ответе (в настоящее время это 1000 сущностей).
3. Общий размер сущностей в ответе превышает максимально допустимый размер ответа (в настоящее время это 4МБ, включая имена свойств, но исключая xml-теги, используемые для REST ).
4. На выполнение запроса требуется больше времени, чем заданный период ожидания сервера (в настоящее время это 60 секунд)

В любом из этих случаев ответ будет включать маркер продолжения в виде специального заголовка. Для запроса к вашим сущностям используются такие заголовки:

* x-ms-continuation-NextPartitionKey
* x-ms-continuation-NextRowKey

Если клиент получил эти значения, он должен передать их со следующим запросом в виде опций HTTP-запроса; во всем остальном запрос остается неизменным. Это обеспечит возвращение следующего набора сущностей, начинающегося с места, обозначенного маркером продолжения.

Последующий запрос выглядит следующим образом:

http://<UriСервиса>/Blogs?<исходныйЗапрос>&NextPartitonKey=<некотороеЗначение>&NextRowKey=<другоеЗначение>

Это повторяется до тех пор, пока клиентом не будет получен ответ без маркера продолжения, что свидетельствует об извлечении всех соответствующих запросу результатов.

Маркер продолжения должен рассматриваться как непрозрачное значение. Оно указывает на точку начала следующего запроса и может не соответствовать фактической сущности в таблице. Если в таблицу добавляется новая сущность, так что Key (новая сущность) > Key (последняя сущность, извлеченная запросом), но Key (новая сущность) < "Маркера продолжения", тогда эта новая сущность не будет возвращена повторным запросом, использующим маркер продолжения. Но новые сущности, добавленные так, что Key(новая сущность) > "Маркера продолжения", войдут в результаты, возвращаемые последующими использующими маркер продолжения запросами.

Теперь рассмотрим модель согласованности, обеспечиваемую Windows Azure Table.

В рамках одной таблицы система обеспечивает гарантии транзакции ACID для всех операций вставки/обновления/удаления для одной сущности.

Для запросов в рамках отдельной секции выполняется изоляция моментального снимка. Запрос обеспечивается согласованным представлением секции с момента его начала и в течение всей транзакции. Моментальный снимок обеспечивает следующее:

1. Отсутствие "грязного считывания". Транзакция не будет видеть незафиксированные изменения, вносимые другими транзакциями, которые выполняются параллельно. Будут представлены только те изменения, которые были завершены до начала выполнения запроса на сервере.
2. Механизм изоляции моментального снимка позволяет производить чтение параллельно с обновлением секции без блокирования этого обновления.

Изоляция моментального снимка поддерживается только в рамках секции и в рамках одного запроса. Система не поддерживает изоляцию моментального снимка для нескольких секций таблицы или других фаз запроса.

Приложения отвечают за сохранение согласованности между множеством таблиц.

В примере MicroBlogging использовалось две таблицы: Channels и Blogs. Приложение выполняет согласование таблиц Channels и Blogs. Например, когда канал удаляется из таблицы Channels, приложение должно удалить соответствующие блоги из таблицы Blogs.

Сбои могут возникать в моменты синхронизации состояния множества таблиц. Приложение должно уметь обрабатывать такие сбои и иметь возможность возобновлять работу с момента, на котором она была прервана.

В предыдущем примере, когда канал удаляется из таблицы каналов, приложение должно также удалить все блоги этого канала из таблицы Blogs. В ходе этого процесса могут возникать сбои приложения. Для обработки таких сбоев приложение может сохранять транзакцию в Windows Azure Queues, что позволяет пользователю возобновить операцию удаления канала и всех его блогов даже в случае сбоя.

Вернемся к примеру с таблицами Channels и Blogs. Channels имеет следующие свойства: Name (Имя) как PartitionKey, пустая строка как RowKey, Owner (Владелец), CreatedOn (Дата создания). И Blogs имеет свойства Channel Name (Имя канала) как PartitionKey, CreatedOn как RowKey, Title (Название), Blog, UserId. Теперь, когда канал удален, необходимо обеспечить, чтобы все ассоциированные с ним блоги также были удалены. Для этого выполняем следующие шаги:

1. Создаем очередь для обеспечения согласованности таблиц, назовем ее "DeleteChannelAndBlogs" (Удаление каналов и блогов).
2. При поступлении запроса на удаление канала от роли веб-интерфейса, ставим в созданную выше очередь элемент, определяющий имя канала.
3. Создаем рабочие роль, которые будут ожидать событие добавления элемента в очередь "DeleteChannelAndBlogs".
4. Рабочая роль изымает элемент из очереди DeleteChannelAndBlogs, задавая для извлеченного элемента очереди время невидимости в течение N секунд. При этом элемент, определяющий имя канала, который должен быть удален, изымается. Если роль работника удаляет элемент очереди в течение этих N секунд, данный элемент будет удален из очереди. Если нет, элемент станет вновь видимым и доступным для использования рабочей ролью. При извлечении элемента рабочая роль делает следующее:
   * В таблице Channels помечает канал как недействительный, чтобы с этого момента никто не мог выполнять чтение из него.
   * Удаляет из таблицы Blogs все записи, для которых PartitionKey = "имени канала", указанному в элементе очереди.
   * Удаляет канал из таблицы Channels.
   * Удаляет элемент из очереди.
   * Возвращается.

Если в ходе выполнения, например, шага 4, возникает какой-либо сбой, производится аварийное завершение рабочего процесса, при этом элемент очереди не удаляется из нее. Таким образом, как только соответствующий элемент очереди станет снова видимым (т.е. когда истечет время, заданное как время ожидания видимости), это сообщение будет вновь извлечено из очереди рабочим процессом, и процесс удаления возобновится с шага 4. Более подробно обработка очередей рассматривается в документации Windows Azure Queue.

### Краткие итоги:

В данной лекции мы ознакомились с одной из абстракций данных Windows Azure Storage - Windows Azure Table. Это технология, которая обеспечивает структурированное хранилище состояний сервиса.

### Ключевые термины

**Windows Azure Table** – абстракция данных, которая обеспечивает структурированное хранилище состояний сервиса.

**Windows Azure Blob** – абстракция данных, которая обеспечивает хранилище больших элементов данных.

**Windows Azure Queue** – абстракция данных, которая обеспечивает диспетчеризацию асинхронных заданий для реализации обмена данными между сервисами.